DBG : A SOURCE LEVEL DEBUGGER FOR C

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

A. SEKAR

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JANUARY, 1988

CERTIFICATE

This is to certify that this thesis entitled "DBG: A SOURCE LEVEL DEBUGGER FOR C" is a report of bonafide work carried out by Mr.A.Sekar, M.Tech. roll number 8511101 under my supervision in partial fulfillment of the requirements for the M.Tech. degree in Computer Science. It has not been submitted elsewhere for a degree.

(G. Barua)

Programme co-ordinator

(V. M. Malhotra)

Thesis supervisor

ACKNOWLEDGEMENTS

I wish to thank my thesis supervisor Dr.V.M.Malhotra for suggesting the project and for egging me on to the completion.

I am greatly indebted to my programme co-ordinator Dr.G.Barua, but for whose timely help this project would have been a failure.

I wish to place on record my gratefulness to all the faculty members of the CSE department, especially Dr.S.Biswas, Dr.R.Sangal, and Dr.R.K.Ghosh, for their consistently fair attitude throughout the programme.

The written word is not powerful enough to express my feelings towards my friends Mr.Vijayan Rajan (CSE) and Mr.R.R.Nagarajan (Met).

A.SEKAR.

ABSTRACT

A source level debugger (DBG) has been built for the high level language C on the Uptron S-32 m/c running the Unix operating system. The existing assembly level debugger ADB has been utilised as a tool. This thesis is a report of the structure and the capabilities of DBG.

CONTENTS

CONTENTS

Chapter 1

INTRODUCTION

One of the major problems in writing a program is ensuring its correctness. *Program provers* are available only for certain classes of algorithms, and program verification techniques are, as yet, primitive. A lot of theoretical work has been done in this area [Loe 1], but, from the point of view of the user, there is no way to prove, in the mathematical sense, the syntactic and semantic correctness of a given program.

In the case of a few programming languages like Pascal, it is possible to ensure complete syntactic correctness during the compilation. The current trend is to perform part of these checks even before this stage with the use of *syntax directed editors* [All 1]. With their use, writing syntactically correct programs has become reasonably simple. The main problem is in ensuring semantic correctness.

The feasible, and universally adopted alternative to program proving is *program testing*. With most programs, the input sample space is infinite, or atleast, the number is sufficiently high to make exhaustive testing impossible. The usual solutions resorted to are representative and limit tests, wherein typical and

limiting values of the inputs are fed to the program, and the outputs produced are checked against hand-computed values.

A program that works the first time it is tried is every programmer's dream. It does happen once in a while, but typically a program initially contains a few *bugs* (mistakes) that are spotted and corrected in a later phase called *debugging*.

Debugging is the process of identifying bugs, and correcting them. It involves identification of inputs for which a given program 'fails' (identifying the code responsible, ~~or~~ and correcting the code. The first of these tasks is an intuitive process, aided solely by some standard guidelines. The last amounts to programming itself. The second task is definitely nontrivial, and a certain amount of help can be offerred by the system itself. The class of programs that offer such help have come to be known as *debuggers*.

Based on whether they operate with the source code of a high level language, or some levels below it (typically assembly code) debuggers can be broadly classified into two categories:

(i) Source level debuggers, also called high level debuggers;

(ii) Assembly level debuggers, or low level debuggers.

Assembly code is typically produced by the compilers of high level languages, and are common to all compiled languages in a given system. Consequently assembly level debuggers are more general, and any given system contains only one of this class. (e.g., DDT on the Dec 1Ø). Source level debuggers are more useful, in that the user need not be aware of many details about the workings of compilers. But this is achieved only at the cost of generality: a source level debugger is language specific, and more often than not, compiler specific too. So a typical system will need to have quite a few source level debuggers, atleast one for each high level language supported by it.

A compromise is usually struck, and bootstrapping techniques are resorted to. That is to say an assembly level debugger is written first, and source level debuggers are written using this. (e.g., PASDDT on the Dec 1Ø system).

The current interest in debuggers is on building debuggers for distributed environments, for optimizing compilers and for integrated environments. For details refer to The Proceedings of the ACM symposium on this issue [ACM 1].

This project is an attempt at developing such a source level debugger for the high level language C [Ker 1], supported by the Unix operating system on the Uptron S-32 system (available in the department) based on the existing assembly level debugger ADB [Bou 1].

Any program is constrained by the environment in which it has to function. In *Chapter 2* of this thesis relevant details about the environment are presented. *Chapter 3* discusses the structure of DBG, the debugger built. Problems encountered, and solutions resorted to are also mentioned in brief. *Chapter 4* summarises the results obtained, their limitations, and possible improvements.

Chapter 2

THE ENVIRONMENT

The Uptron S-32 system is a M68000 based m/c running the Unix System III operating system. It supports the high level language C, which essentially means that it has got a compiler for the language C. A debugger being a runtime tool, the amount of source-level information it can handle is usually determined by the amount of information the compiler retains in its outputs. Hence it is necessary to consider the compiler, especially its outputs and this is done in Section 1 of this chapter. Since the debugger has been built on top of the existing low-level debugger ADB, its capabilities are again a limiting constraint. The details are presented in Section 2. The Unix operating system provides an environment in which system programs like debuggers can be easily developed. Section 3 deals with certain of these aspects. Two specific tools Lex and Yacc that have been used in this project are also discussed in the same section.

2.1 THE COMPILER

The Uptron S-32 system supports the Portable C Compiler developed at the Bell Laboratories [Joh 1]. From the user point of view, it consists of three

phases: Preprocessing, Generating the assembly code, and generating the executable code. When invoked as "/lib/cpp <filename>", it performs the first of these three, and exits. When called as "cc -S <filename>", the first two phases are executed and the assembly code is produced. Called as "cc <filename>", all three phases are executed in turn. The other existing phases are not relevant to the present discussion.

During its execution the compiler creates a structure called 'Symbol Table' where details of variable names, their types, addresses etc., are stored. By default most of this information is lost upon completion of the compilation. Typically options can be specified so that this information is retained. Unfortunately the current implementation does not support such options, and hence, this information is not available to runtime routines. The only way to access this information is to imitate the compiler, a solution we resort to, as will be explained in Chapter 3. However, the fact that the imitation is done necessiates a discussion on the address allocation schemes themslves [Uni 1].

Program Parts

When a C program is compiled and assembled, the program is split into three parts. They are:

1. The executable code of the program, called the text area;

2. The initialised data area that contains constants, strings and so on;

3. The uninitialised data area or the bss area.

These three parts appear in the above order. The compiler/assembler combination produces the first two. The loader generates the third at load time.

Data Representations

All data elements are stored such that their least significant bit is in the highest addressed byte, and their most significant bit is in the lowest.

The sizes of the basic objects are given in the following table:

type	no. of bits	aligned on
Char	8	byte
Short	16	word
Long	32	word
Int	32	word
Float	32	word
Double	64	word
Pointer	32	word

Arrays:

The base address is aligned on word boundary. Elements are stored in contiguous locations in row major order. If size of the last dimension is omitted, then allocation is made for array of pointers.

Structures:

The elements of a structure are not reordered. Elements are assigned space contiguously, each of the elements aligned on a word boundary.

Globals:

Globals are, by default, placed in the bss area. When there is an initialization in the declaration itself, they are placed in the data area.

Static Variables:

Static variables, initializd or otherwise, are allocated space in the data area.

Automatic variables:

These are allocated space on the stack dynamically. During the execution of a function, the stack has the following structure:

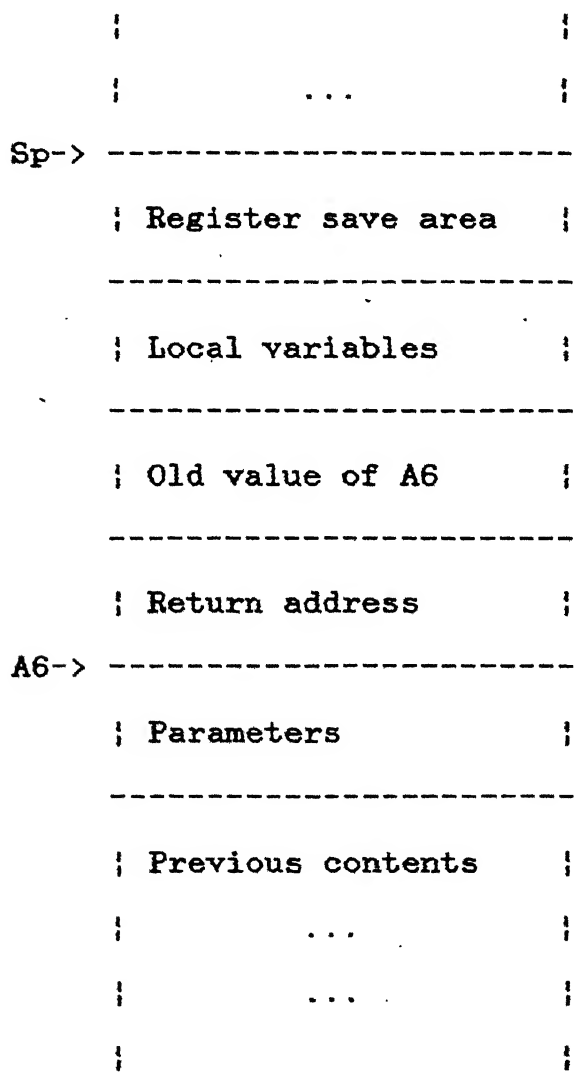


fig.1: Contents of the stack

The Assembly Code

The discussion here is restricted to the details that have a direct relevance. A comment is placed on a new line with the prefix "¦ ". Labels are declared by the statement ".globl <label>" and are used as "<label>: <stmt>".

The assembly code produced by the C compiler contains the assembly code corresponding to the source line <ln> between the comments "¦l <ln>" and "¦e <ln>". If the code is split into more than one part, then each of the parts is prefixed and postfixed in this manner.

2.2 ADB

ADB [Bou 1] is a symbolic debugging tool available on Unix. A survey of some of its features follows.

Invocation and Exiting

ADB is invoked as "adb <objfile> <corefile>" where <objfile> is an executable Unix file and <corefile> is a core image file produced on executing an error. The typical invocation is a "adb a.out core". To

exit from ADB the request "$q" or "$Q" must be used.

Breakpoint Control

Break points can be set only at function entry points. The request to place a breakpoint is "<address> :b". To delete a breakpoint that has already been placed, the request "<address> :d" is used. In these, <address> is the name of a function.

To print all the breakpoints that have been placed, the command "$b" is used.

To start the execution the command is ":r". The execution continues till a breakpoint is encountered or the program terminates. To continue execution from a breakpoint, the command is ":c".

General Request

The general form of requests to ADB is <address><count><command><mod>. The relevant commands are "?" to print contents from a.out file, and "!" to escape to shell.

Printing Values

The modifier "=" followed by an optional format specification prints the address of the variable named.

The monadic operator "&" can be used to print the actu al values. The formats are "b" to print one byte in oc tal, "d" to print one word in decimal and "f" to prir one word in floating point.

The command "$r" prints all the register values. "< a6" gets the contents of the register A6 which is the frame pointer.

The command "= <string>" prints <string> as its output.

The default format is hexadecimal. Decimal numbers carry the prefix "Ød" and hexadecimals "Øx".

Register variables can not be handled by ADB.

2.3 PROCESSES AND THE REST

The Unix Operating System [Bou 2] provides an environment conducive to the development of system programs. Some of the relevant features are discussed in this section.

A new process (child) can be created by a process (parent) using the system call *"fork"*. An existing process will exit when its execution is complete, or when it receives a *"kill"* signal.

Communication between processes is facilitated by the system call *"pipe"*. A pipe is a FIFO (First In First Out) file. A process attempting to read from an open pipe will wait till input is available. Files can be opened for reading and writing. The latter can be in the overwrite mode or in the append mode. *"fopen"* and *"fdopen"* are the relevant calls.

Input and Output of a process can be redirected by its parent to a file at the child's invocation time by "<process>'<'<infile>'>'<outfile>" where <infile> and <outfile> are the input and output files respectively.

A process can redirect its own input and output to files using *"freopen"* or *"dup"*.

A process can direct a new process to be overlayed over itself by using the *"exec"* system call.

A new shell command can be executed by using the *"system"* subroutine.

All the above mentioned calls can be directly used by C programs.

Lex [Les 1] is a lexical analyser generator that is especially useful as a preprocessor for *Yacc*

(Yet Another Compiler-Compiler) [Joh 2]. Yacc is a LALR parser generator [Aho 1] that helps rapid development of parsers [Sch 1]. Both Lex and Yacc can generate C code as their output. Besides, both have provisions for containing C routines embedded in their input. Both these packages have been extensively used in this project.

Chapter 3

THE SYSTEM: DBG

3.1 THE APPROACH

DBG accepts C source code as its input. It itself compiles the program, and executes it, if the user issues the command for that. The object code generated by compiling the user program is passed on to ADB as its input. As a rule, when something can be done using ADB, it will be done using ADB. DBG extracts information from the source code only when such information is not otherwise available to ADB.

Additional information is used in two ways: In breakpoint control (where the line number information is needed) it is incorporated into the generated object code of the user program itself; In printing of variable values (where type and address information is needed) it is extracted from the surce code, and is used directly by DBG.

The structure of the program is as follows: The *main program*, initially called the *preprocessor* creates a child process named *symbol table process* and invokes it. The symbol table process extracts the type and address information from the user source code, writes it in a temporary file called *sym.out* and exits.

Concurrently, the preprocessor generates the assembly code for the user source code, introduces line number labels in it, generates the object code, and waits for the symbol table process to exit. When that happens, the preprocessor obtains the symbol table information from the file sym.out and invokes the *control program* phase of the main program.

The control program creates two new processes called the *adb process* and the *display process*. The adb process then invokes ADB. From this point onwards the main program is called the *Command Interpreter* (CI). The display process essentially routes the outputs of the adb process to the terminal and the CI.

Debugging commands from the standard input (keyboard) are accepted by the CI and are checked for syntactic correctness, and, to a limited extent, for semantic correctness. Depending on the command, a sequence of sub_commands are issued to ADB through a pipe named *adb_in*. Outputs of ADB are written on the pipe *adb_out*, from where they are read by the display process. Analysing the information received, the display process may write it on the terminal, send it to the CI through the pipe *ret_val* or simply gobble it up.

When the user indicates the termination of a debugging session, the parent process performs the cleaning operations like deleting the temporary files, killing the processes created etc., and exits. Sessions are distinct, in that no information extracted during a session is retained for the possible subsequent sessions.

The individual components of the system are examined in the following sections on a process-by-process basis. Adb process is not explicitely discussed, because, it does very little more than invoking ADB.

3.2 THE MAIN PROGRAM

3.2.1 The Preprocessor

The preprocessor essentially extracts the additional information (specified earlier) needed, and stores them in structures that facilitate easy access by the later phase Command Interpreter. The structure of this phase of the main program is shown in figure 2.

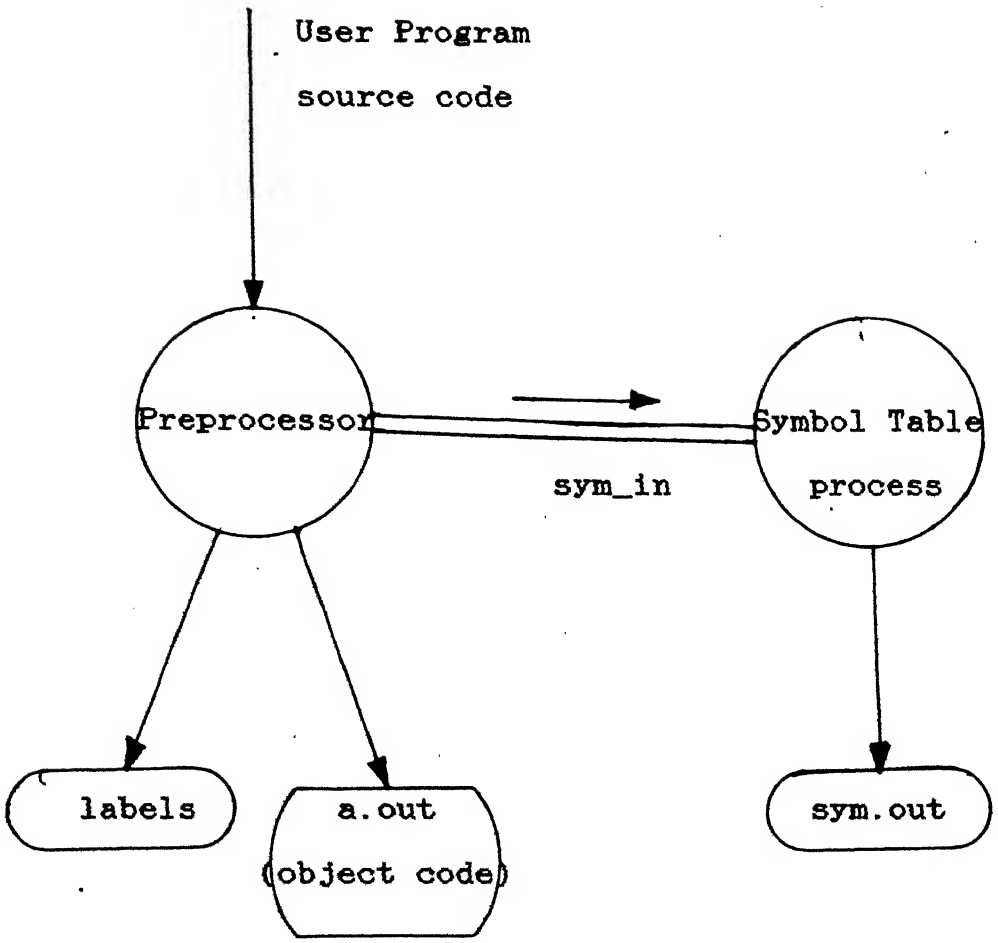


fig.2: The Preprocessor

The individual actions performed are as follows:

First the smbol table process is created. The user program is C-preprocessed and is written on the pipe sym_in. Then the C compiler is called with the -S option to create the assembly code. The assembly code generated contains, as comments, the source line

numbers. Hence the mapping between the C source code and the corresponding assembly code is preserved by the C compiler itself. These comments are located, and unique labels and the necessary declarations are introduced.

Any label thus introduced contains the following four fields:

(i) The prefix "_" (underscore). The selection is due to the fact that ADB breaks at function entry points by trapping the labels, and the function names in the compiler generated assembly code carry the same prefix. Functionally, the idea is to "fool" ADB to trap the line number labels as function entry points.

(ii) The prefix "zz" used to make the label distinct from typical user-defined labels.

(iii) A single alphabet. This is used to differentiate between the breakpoints of different files in case the user has his program spread over many files. The alphabet is different for different files.

(iv) The line number within a given file. This

information is extracted from the comment itself.

The labels are also entered into a temporary file *"labels"* to enable semantic checks during the CI phase.

The preprocessor next calls the C compiler with the modified assembly code as input to generate the object code file *"a.out"*.

Then the program halts till the symbol table process signals its exit. Upon receiving this signal, the information in the file sym.out is read, and is stored in suitable structures after making suitable modifications. The structures are identical to those used by the symbol table process, and hence are described in section 4. The necessity and the nature of the modifications are also indicated in the same place.

3.2.2 Control Program

The function of the control program phase is to create the child processes and to set up the cmmunication between them. The result of its execution is shown in figure 3.

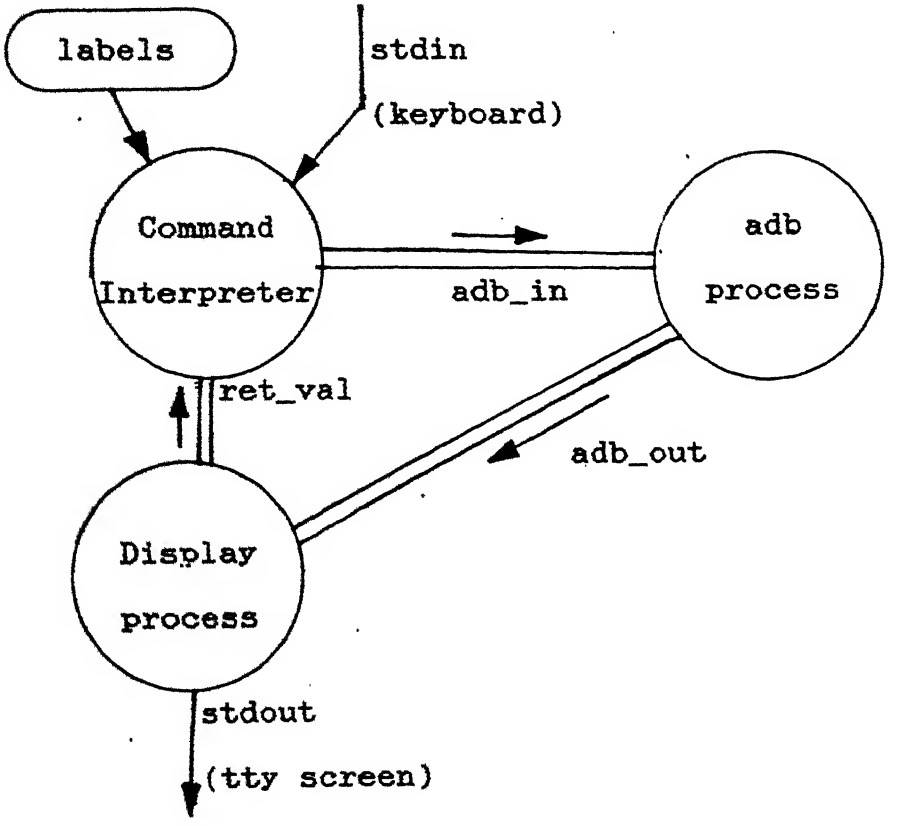


fig. 3: control program

In the figure, adb_in, adb_out and ret_val are the names of the pipes, and the arrows indicate the direction of flow of the information: The process at the tail writes into the pipe, and the process at the head reads from the pipe. Essentially, the control program

creates two child processes adb process and display process, and invokes them after making the necessary files and pipes available for reading/writing.

3.2.3 Command Interpreter

The command interpreter accepts the debugging commands issued by the user, and controls the consequent actions performed by DBG. It has been written using the packages Lex and Yacc.

The syntax check is performed on the user command by Yacc itself. The semantic ckecks performed are of two kinds: When a breakpoint is specified on a line, a check is performed, using the file labels, as to whether the breakpoint is possible. If not, suggestions are given about where one can be placed. When a print command is isued, a check is performed as to whether the appropriate declaration is present in the user program. If not, suitable diagnostic messages are issued.

Instead of describing the actions performed for the individual commands, we describe this part of the system in terms of its functions.

Breakpoint Control:

By breakpoint control we mean the part of the system (DBG) which handles the following tasks:

(i) Setting a breakpoint: i.e., specifying the suspension of the execution of the user program at a user-specified location (line number).

(ii) Removing a breakpoint: i.e., specifying that the execution need not be suspended at a user-specified location. This being the default action, it is meaningful only if a breakpoint has been set at this location previously.

(iii) Start/restart execution: i.e., begin the execution of the user program. Since there is no facility for "backward execution", the effect is partly achieved by terminating the current run (of the user program) and starting the next one.

(iv) Break at a breakpoint: i.e., suspending the execution on encountering a breakpoint.

(v) Break on error: i.e., suspending the execu-

tion of the user program on runtime errors.

Some of these are already performed by ADB. In such cases, a predefined command is issued to ADB. For others, suitable processing needs to be done.

Printing Values:

When the execution of the user program gets suspended, the user can print the values of variables available to him at that point.

To print the value of a variable in the correct format, it is necessary to know the address of the variable, and its type. In the case of global symbols, the address information is already available to ADB, and hence, a simple request to ADB will suffice. In the case of automatic variables, a request to ADB will get the frame pointer, and the symbol table contains the offset from it, and so, from these two, the address of the variable can be computed. In the case of simple variables, a command is issued to ADB to get the value, and the result is converted in to the proper format and printed.

DBG also provides facilities for more involved printing. If flag is a variable, then DBG permits ac-

cessing *flag, flag[5], flag.elem etc., provided flag has the correct declaration necessary to perform these operations. The series of such operations can also be performed. In all such cases, a series of commands have to be issued to ADB before the value can be printed. In short, this part of the system is an evaluator for simple expression.

Miscellaneous commands

DBG also offers the following facilities: Getting the C backtrace, Listing out the breakpoints that are currently active, Printing source files, Executing shell commands, and Using the ADB directly. There should not be any need to use the last of these - it has been provided only for the sake of completeness.

3.3 DISPLAY PROCESS

The display process is essentially a routing program. Anytime the command interpreter sends requests to ADB, it also sends a signal to the display process (through ADB itself). The signal is an instruction as to what must be done with the output that is produced by ADB. The choices are passing it on to the user by printing it on the terminal, sending it to the command

interpreter through the pipe ret_val, and simply gobbling it up. In the first two cases, a certain amount of filtering also needs to be done.

The display process has a difficulty in sensing the end of the output produced by ADB. To elliminate this problem, the command interpreter sends another signal to the display process immediatelly after sending a command to ADB.

The display process is essentially controlled by these two signals that ensure synchronisation between the three processes involved.

3.4 SYMBOL TABLE PROCESS

The symbol table process is created and invoked by the preprocessor. This part of the system has also been written using the packages Lex and Yacc. Its main task is to construct the symbol table by parsing the C-preprocessed user program.

The symbol table constructed is a linked list of *fnnodes*. All the declarations of a given function are associated with the fnnode corresponding to this function. Uniformity is achieved by considering the global variables to be the local variables of the function

"global".

Fnnode has the simple structure given below:

```
struct fnnode
        {
        char    fnname [ IDMAX ];
        struct  idnode  *paramp;
        struct  idnode  *localp;
        struct  fnnode  *nxtfnp;
        };
```

It contains four elements. The first, fnname, is the name of the function. The second element paramp is a pointer to the head of the linked list of formal parameters. The element localp is a pointer to the head of the linked list of local variables of this function. The last element nxtfnp is a pointer to the next node of the linked list of fnnodes.

Information about an identifier that is not a function name is stored in a different type of structure called *idnode*. Local variables, parameters, globals, typedef names, structure and union names, their components, details about arrays, and pointers are all stored in idnodes. With the exception of fnnodes, all the information in the symbol table is stored in the idnodes. The uniformity thus achieved gives rise to a

certain amount of space inefficiency, but the gains in terms of ease of programming and clarity are enormous.

Idnodes have the following structure:

```
struct idnode
        {
        char    idname  [ IDMAX ];
        int     sc;
        int     idtype;
        struct  idnode  *eltypep;
        int     elnum;
        int     idsize;
        int     idaddr;
        struct  idnode  *nxtidp;
        };
```

The first element, idname, is the name of the identifier. Its storage class (Auto, Global, Type, Struct etc.,) is recorded in sc. The type (Int, Float, Struct, Pointer etc.,) of the identifier is stored in idtype. Eltypep is a pointer to the elements. This field is discussed in detail in a sort while. Elnum contains the number of elements that are present, used only if the identifier is an array. The size of the object under discussion is stored in idsize, and its relative address from the base address of the current function in

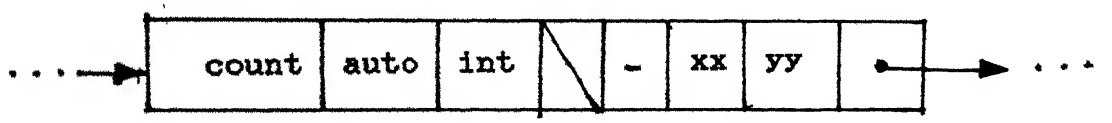
idaddr. The last field nxtidp is a pointer to the next identifier encountered *at the same level.*

The structure of the symbol table constructed is more explicitely explained by discussing the various kinds of objects, and the nodes created. Such a discussion follows.

Simple Variables

A simple declaration is stored in a single idnode in a straightforward manner, as illustrated by the following example:

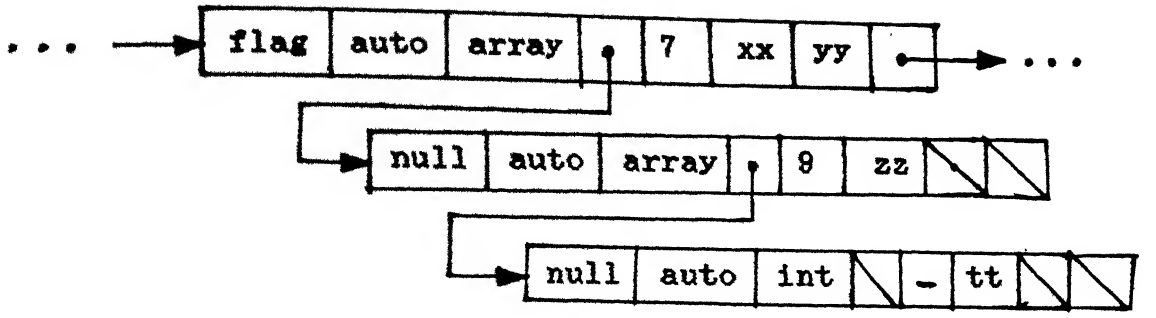
Declaration: auto int count;



Arrays

In the case of arrays, atleast two idnodes are used, the parent specifying it to be an array, and the child(ren), with a special dummy head "null" containing the details about its elements.

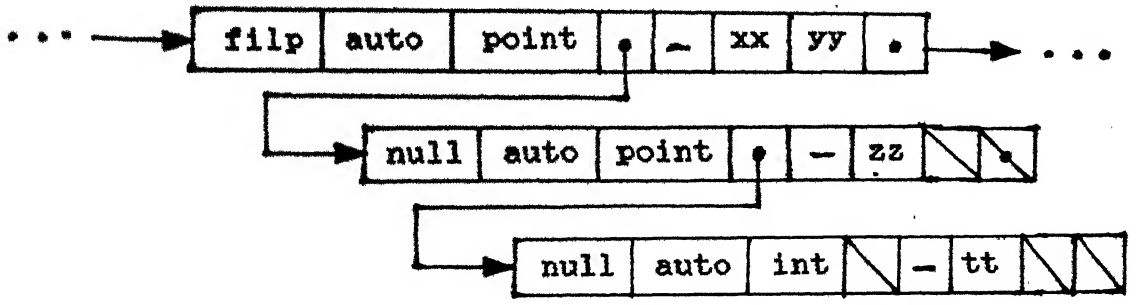
Declaration: auto int flag[7][9];



Pointers

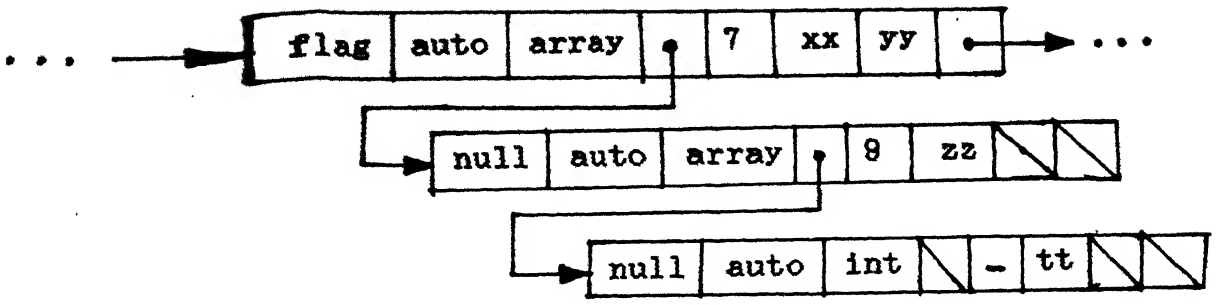
Pointers are stored in a manner similar to the arrays except that the elnum field is not used. Arrays without subscripts are also treated as pointers, in conformance with the semantics of C.

*Declaration: auto int **filp;*



Type names

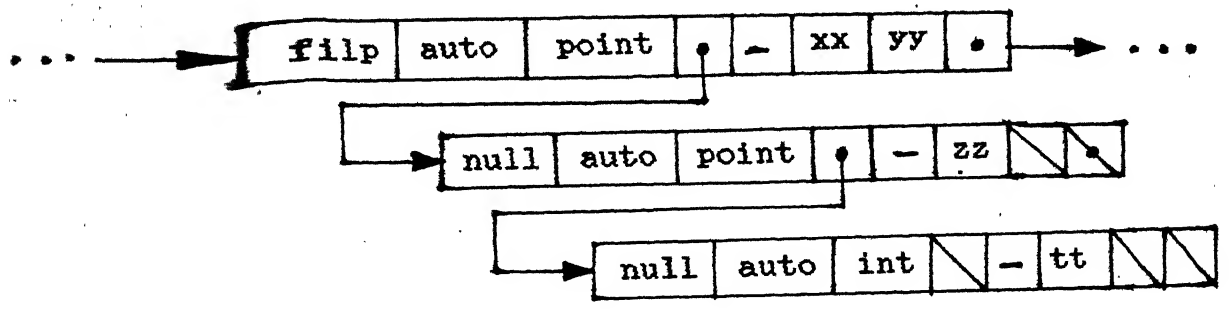
These are stored just like variables, except that the sc field specifies them to be types, and the address field contains (-1) to indicate that space must not be allocated.



Pointers

Pointers are stored in a manner similar to the arrays except that the elnum field is not used. Arrays without subscripts are also treated as pointers, in conformance with the semantics of C.

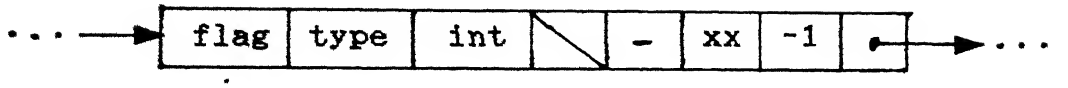
*Declaration: auto int **filp;*



Type names

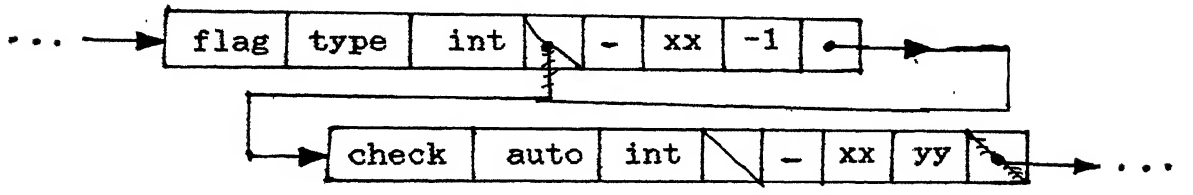
These are stored just like variables, except that the sc field specifies them to be types, and the address field contains (-1) to indicate that space must not be allocated.

Declaration: typedef int flag;



When type names are used in a declaration, idtype, idsize, elnum, and eltypep fields are copied.

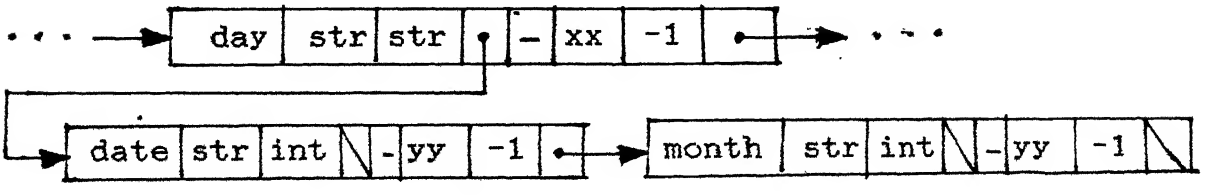
Declaration: typedef int flag; flag check;



Structures and Unions

The elements of a structure/union are kept in a linked list of identifiers. The head is stored in eltypep of the idnode containing the name of the structure. As in the case of type names, the declaration contains (-1) in the idaddr field.

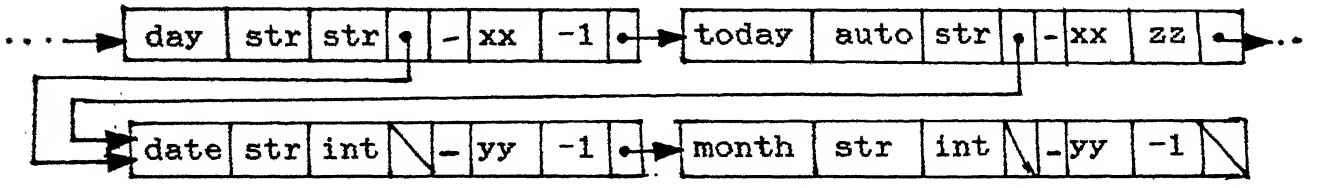
Declaration: struct day { int date; int month;};



When structure or union names are used in a declaration, idtype, idsize, elnum, and eltypep fields are copied.

Declaration: struct day { int date; int month;};

struct day today;



Using the structures and mechanisms described above, the symbol table is constructed by the symbol table process.

The entire table is now copied to the preprocessor process. For this purpose, a search is performed on the symbol table, and all non-null pointer variable values are altered to relative addresses, relative to the head node of the symbol table (the dummy fnnode "global"). The symbol table is then written on to the file sym.out as linear space. The preprocessor, at the time it reads the file sym.out, converts all relative addresses to absolute addresses using the base address

of the first node in that process. This copying is possible because care has been taken to ensure that the allocation of space in both these processes is contiguous.

After writing out the symbol table into sym.out, the symbol table process exits, which action signals the preprocessor to continue as discussed in section 2 of this chapter.

Chapter 4

CONCLUSION

A debugger has been built, in accordance with the specifications stated in chapter 1. Any system can be improved by additional efforts. Possible improvements have been hinted during the discussion itself, but, for the sake of completeness, they have been explicitely stated below.

The entire C language is not being recognized by the current version. Fields, function pointers, and declarations in the inner blocks are some of the cases that have been omitted.

In the command interpreter part, printing of entire structures or arrays, given the name, can be done. Again, parameter values can be accessed. These two also have not been done, though they can be included in the next version without any major changes to the structure of the system.

But there are other changes which will need substantial rewriting. For example, register variables can not be handled by ADB, and hence, to include them will mean bypassing ADB, and hence the entire system will have to be redesigned. Assigning values to variables

midway through execution, useful for debugging later program parts before the earlier parts are complete, can not be done due to the same reason.

The basic problems, though, are not the limitations of ADB, but are the limitations of the C compiler itself. Writing an efficient debugger that works with a compiler that does not lend itself to the job in hand, creates too many problems, not all of which are solvable. Writing a new C compiler that can retain all the information that will be needed by debuggers is definitely a worthwhile and feasible project. With such a compiler around, writing a debugger with the specifications of DBX (available on the HCL systems in the department) will not be an impossible task.

REFERENCES

[ACM 1] *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging 1983.*

[Aho 1] *Aho, A. V., and Ullman, J. D., Principles of Compiler Construction, 1977, Addison-Wesley.*

[All 1] *Allison, L., Syntax Directed Program Editing, Software-Practice and Experience, Vol 13, No. 5 (1983).*

[Bou 1] *Bourne, S. R., and Maranzano, J. F., A Tutorial Introduction to ADB, Uniplus Manual Volume II.*

[Bou 2] *Bourne, S. R., The Unix System, 1982, Addison - Wesley.*

[Joh 1] *Johnson, S. C., A tour Through the Portable C Compiler, Uniplus Manual Volume II.*

[Joh 2] *Johnson, S. C., Yacc: Yet Another Compiler-Compiler, Uniplus Manual Vol II.*

[Ker 1] *Kernighan, B. W., and Ritchie, D. M., The C Programming Language, 1978, Prentice-Hall.*

[Les 1] *Lesk, M. E., and Schmidt, E., Lex: A Lexical Analyzer Generator, Uniplus Manual Volume II.*

[Loe 1] *Loeckx, J., and Sieber, K., The Foundations of Program Verification, 1984, John Wiley.*

[Sch 1] *Scheiner, A. T., Introduction to Compiler Construction with Unix, 1985, Prentice-Hall.*

[Uni 1] *"C" Interface Notes for 68000 Unix Systems, Uniplus Manual Volume II.*

Appendix I

COMMAND SUMMARY

Conventions

<line> = Source code file line number.

<file> = Source code file name.

<var> = Variable name.

<func> = Function name.

aa <b> = The string "aa" followed by <b>.

{<b>} = Optional use of <b>.

{a <b> ...} = Zero or more occurances of "a <b>".

brk

Uses

The command "brk" is used to set breakpoints at specified source code line numbers.

Syntax

brk {<file> : } <line> { , {<file> : } <line> ... }

Diagnostics

ok = The breakpoint is possible, and has been set.

Error: Not possible: Try <line>.

Error: Not possible: Not that many source lines.

Error: Unknown file name = <file> is not a parameter in the invocation.

Examples

brk 23

brk 23, 24

brk try.c:23

brow

Alias

disp

Uses

This command is used to look up source files. The source file line numbers are also displayed. The display contains the necessary commands to display any

source line.

Syntax

brow <file>

Example

brow try.c

btr

Uses

This command is used to get the C backtrace of the running program.

Syntax

btr

exit

Alias

quit

Uses

Used to exit from DBG.

nxt

Uses

Once a breakpoint has been encountered, this command is used to continue the execution.

Diagnostics

no process = No suspended process under the control of DBG.

pa

Uses

This command is used to print the values of automatic variables.

Syntax pa <func> / <var>

pa <func> / <var> [<constant>]

pa <func> / <var> . <var>

pa <func> / * <var>

Diagnostics

Error: Function Unknown = Declaration of <func> not present.

Error: Variable Unknown = Declaration of <func> absent in <func>.

Examples

pa main/flag

pg

Uses

This command is used to print the values of global variables.

Syntax

pg <var>

pg <var> [<constant>]

pg <var> . <var>

pg * <var>

diagnostics

Same as in "pa".

rmv

Uses

This command is used to delete a breakpoint that has been set.

Syntax

Same as in "brk".

run

Uses

Starts the execution of a.out.

zlist

Uses

Used to get the list of all active breakpoints.

Appendix II

EXAMPLES

The program

```
/* program in file simple.c */

int sum;

main()
        {
        int count;

        count = Ø;
        while (count < 3)
                {
                sum += 2;
                count += 1;
                }
        }
```

The Session

```
unix> dbg simple.c
wait.......
ready
dbg> brk 13
ok
```

```
dbg> zlist
file simple.c line 13
dbg> run
a.out running
break at: file simple.c line 13
dbg> pa main/count
count = Ø
dbg> pg sum
sum = 2
dbg> nxt
break at: file simple.c line 13
dbg> pa main/count
count = 1
dbg> pg sum
sum = 4
dbg> rmv 13
ok
dbg> zlist
dbg> nxt
a.out terminated
dbg> exit
unix>
```